Énoncé de projet 2 : Starlight

R. Absil, M. Bastreghi, J. Beleho, A. Rousseau,
N. Vansteenkiste et M. Wahid

13 mars 2015

Résumé

Ce document détaille l'énoncé de votre projet 2 des laboratoires C++. Celui-ci consiste à implémenter une interface graphique permettant de jouer à *Starlight*, un puzzle en deux dimensions basé sur la lumière.

1 Introduction

Starlight est un petit jeu en deux dimensions se jouant sur une carte rectangulaire, comportant une source de lumière, émettant un rayon rectiligne. Le but du jeu est d'atteindre une cible avec ledit rayon, en évitant les obstacles via notamment des miroirs réfléchissant la lumière.

Le but de ce projet est de vous proposer une application ludique à implémenter graphiquement, sur base d'un squelette de classes fourni. Mis à part quelques parties algorithmiques relativement simples, vous ne devez donc que créer une interface graphique pour ce projet.

Ce document est structuré comme suit : en premier lieu, la Section 2 vous décrit formellement les règles du jeu, ainsi que tous ses composants. Sur base de cela, la Section 3 vous décrit sommairement les classes qui vous sont proposées comme squelette pour votre projet. La Section 4 poursuit en détaillant le format de sauvegarde des cartes utilisé dans le jeu. Ensuite, la Section 5 détaille des consignes particulières relatives aux exigences liées à ce projet. La Section 6 propose plusieurs points de bonus à éventuellement améliorer une fois les objectifs principaux remplis. Par après, la Section 7 rappelle à l'étudiant intéressé, entre autres, quelques rudiments de trigonométrie et de géométrie qui seront utiles à l'implémentation du projet. Enfin, la Section 8 conclut ce document. Les fichiers d'en-têtes d'une solution partielle au problème ici posé sont fournis en Annexe A.

2 Règles du jeu

Starlight est un puzzle à deux dimensions se jouant sur une carte rectangulaire. Le but du jeu est de dévier un rayon lumineux d'une source vers une cible en évitant certains obstacles. Plus particulièrement, on trouve les éléments suivants sur une carte.

- Une unique *source* : cet élément émet un rayon lumineux d'une longueur d'onde donnée sous un certain angle.
- Une unique *cible* (ou *destination*) : cet élément doit être éclairé par un rayon lumineux pour remporter la partie.
- Un ensemble de *miroirs* : un miroir est un objet réfléchissant la lumière d'un seul côté suivant le schéma naturel de la réflexion de la lumière. Plus particulièrement, un rayon incident à un miroir sous un angle θ_i sera réfléchi sous le même angle θ_r, comme illustré [1] à la Figure 1.
- Un ensemble de *murs* : les murs ne réfléchissent pas la lumière. Tout rayon incident à un mur ne se propage pas, et « s'arrête » donc là où il y est incident.
- Un ensemble de *lentilles*. Les lentilles sont des objets transparents qui ne laissent passer un rayon lumineux que dans un certain intervalle de longueur d'onde $[m, n]$. Si un rayon lumineux possède une longueur d'onde ν telle que $m \leqslant \nu \leqslant n$, il traverse la lentille sans subir aucune modification. Sinon, la lentille stoppe le rayon (elle se comporte comme un mur).
- Un ensemble de *cristaux* : un cristal est un élément transparent qui modifie la longueur d'onde d'un rayon, en l'augmentant ou la diminuant. Tout rayon qui traverse un cristal le traverse donc sans subir de modification de trajectoire, mais voit sa longueur d'onde modifiée.
- Un ensemble de *bombes*. Les bombes sont des objets qui, si éclairés, explosent et font automatiquement perdre la partie au joueur.
- Un ensemble de *rayons*. Initialement émis par la source du jeu, ils sont rectilignes et se réfléchissent sur les miroirs. Un rayon est donc un segment de droite. Sur la Figure 1, on voit donc deux rayons, le rayon P et le rayon Q. Un rayon possède également une autre caractéristique : sa longueur d'onde. La longueur d'onde d'un rayon permet de déterminer, comme mentionné ci-dessus, si oui ou non un rayon traverse une lentille. Elle est modifiée par un cristal.

Tous les objets ci-dessus sont immobiles, à l'exception des miroirs qui peuvent être déplacés et pivotés dans certaines limites (cf. Section 3). Les rayons lumineux ne sont présents dans le jeu que lorsque la source lumineuse est allumée.

1. Domaine public

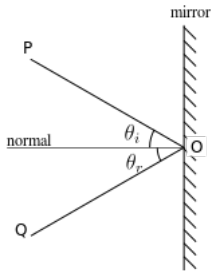

FIGURE 1 – Illustration de la réflexion de la lumière

3 Squelette de classes et système de coordonées

Cette section décrit sommairement ce squelette ; pour des informations plus détaillées, vous êtes encouragés à consulter la documentation du projet, rédigée pour `Doxygen` [4]. Vous trouverez également dans cette section une description précise du système de coordonnées à utiliser pour votre projet.

Le squelette de classes suivant vous est proposé afin de faciliter votre travail de modélisation. Vous êtes libres de l'utiliser comme bon vous semble sous réserve que vous compreniez ce qui y est programmé. Vous pouvez également recoder l'intégralité du projet, sur base de justifications pertinentes.

Le squelette de classes, réunies dans des couples de fichiers de déclaration header `.h` et de définition `.cpp`, est le suivant.

- Classe `Point` : modélise les points à deux coordonnées entières, utilisés pour décrire, entre autres, la position des objets présents sur la carte.
- Classe `Source` : modélise la source. La source est un objet carré qui émet un rayon lumineux sous un certain angle. Le rayon lumineux est émis depuis la position de la source, et possède une longueur d'onde initiale.
- Classe `Dest` : modélise la cible. La cible est un objet carré qui, si illuminé, fait remporter la partie au joueur.
- Classe `Nuke` : modélise les bombes. Une bombe est un objet circulaire qui, si illuminé, fait perdre la partie au joueur.
- Classe `Wall` : modélise les murs. Un mur est un simple segment de droite qui ne réfléchit pas la lumière.
- Classe `Crystal` : modélise les cristaux. Un cristal est un objet circulaire qui modifie la longueur d'onde des rayons lumineux qui le traversent sans les dévier.
- Classe `Lens` : modélise les lentilles. Une lentille est objet elliptique qui ne laisse passer que les rayons dans un intervalle de longueur d'onde

donnée, sans modifier leur trajectoire.

- Classe `Mirror` : modélise les miroirs. Un miroir est un segment de droite qui ne réfléchit la lumière que d'un seul côté (si la lumière arrive du « mauvais » côté, le miroir se comporte comme un mur). Les miroirs peuvent être déplacés et pivotés dans une certaine limite. Les miroirs possèdent donc quatre caractéristiques majeures : un pivot décrivant la position du miroir, une longueur, `xpad`, un entier décrivant la position du pivot sur le miroir, et un angle décrivant l'inclinaison du miroir par rapport à l'horizontale. Ces caractéristiques sont illustrées à la Figure 2. Sur ce dessin, la lumière n'est réfléchie que du côté « non hachuré » du miroir. Le point P décrit le pivot du miroir, α l'inclinaison du miroir, *en radians* [3], par rapport à l'horizontale, et `length` la longueur du miroir. Les miroirs sont les seuls objets du plateau à pouvoir être déplacés et pivotés, dans certaines limites décrites par `xMin, xMax, yMin, yMax, alphaMin, alphaMax`. Toute tentative de déplacer ou pivoter un miroir en dehors de ces bornes laisse ses caractéristiques inchangées.
- Classe `Ray` : modélise les rayons lumineux. Un rayon lumineux est un segment de droite muni d'une longueur d'onde, comprise entre les constantes `WL_MIN` et `WL_MAX`. Ces constantes correspondent au spectre de la lumière visible, en nanomètres (nm). Une tentative de modification de la longueur d'onde en dehors de ces bornes la laisse inchangée.
- Classe `Level` : modélise la carte. Cette carte est simplement constituée d'un ensemble des éléments décrits ci-dessus. Cette classe possède également la méthode `computeRays` que vous devez implémenter obligatoirement pour calculer les rayons lumineux quand la source est allumée. Lors de l'instanciation d'un objet de cette classe, les quatre murs correspondant aux bords de la carte sont automatiquement ajoutés dans la carte.

Vous trouverez plus d'informations relatives à ces classes dans la documentation du programme.

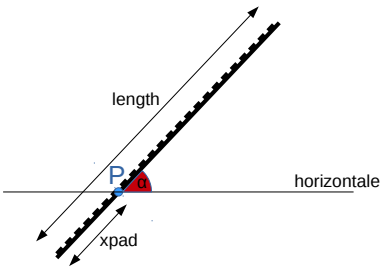

FIGURE 2 – Illustration des caractéristiques d'un miroir

Système de coordonnées

Le système de coordonnées à utiliser est celui classiquement utilisé par les systèmes de rendu de fenêtres :

1. l'origine du repère se trouve dans le coin supérieur gauche de la fenêtre considérée,
2. les coordonnées considérées sont entières : une unité pour un pixel,
3. l'axe *ox* est placé à l'horizontale et est orienté de la gauche vers la droite,
4. l'axe *oy* est placé à la verticale est est orienté du haut vers le bas.

Vous noterez donc que la direction de l'axe *oy* est inversée par rapport à ce que vous utilisez habituellement en cours de mathématiques.

Pour les objets carrés et rectangulaires, la position de l'objet est celle du coin supérieur gauche du carré ou du rectangle considéré, pas son centre. Pour les objets circulaires, la coordonnée de l'objet est celle du centre de l'objet considéré. Pour les objets elliptiques, la position dénote le coin supérieur gauche du rectangle minimal englobant l'ellipse [2], et la largeur et la hauteur dénotent la largeur et hauteur du rectangle englobant.

Seuls les miroirs et les rayons ne sont pas toujours horizontaux ou verticaux. Les murs, la source, les lentilles, etc. c'est-à-dire les carrés, ellipses et rectangles ont leurs côtés ou leurs axes parallèles aux axes *ox* et *oy*.

Les amplitudes des angles fournies dans les fichiers de cartes sont en radians, pas en degrés. Ces amplitudes sont considérées par rapport à l'horizontale, à droite (cf. zéro du cercle trigonométrique [3]).

4 Format des sauvegardes de cartes

Les cartes à charger et sauvegarder sont enregistrés dans un fichier texte d'extension `.map`. Ce fichier texte stocke un élément de la carte par ligne. Vous pouvez supposer que les fichiers texte sont sans erreur [4]

2. Cette convention pour ces objets peut sembler contre-intuitive, mais vous verrez qu'elle simplifie grandement l'usage des API graphiques de nombreuses librairies, `Qt` incluse

3. `http://fr.wikipedia.org/wiki/Cercle_trigonométrique` - Consulté le 13 Février 2015.

4. Et donc, vous ne devez pas vérifier si les caractéristiques fournies pour un élément sont pertinentes (par exemple, bon nombre fourni, les chiffres sont correctement formés, etc.).

La première ligne d'un fichier de carte contient toujours deux nombres entiers positifs, décrivant respectivement la largeur et la hauteur de la carte. Comme le carte est rectangulaire ces données vous permettent de placer les murs qui la limitent. Chacune des autres lignes décrit un composant.

La Table 1 illustre le format de chacun de ces composants. La colonne 1 dénote le type d'élément, la colonne 2 les caractéristiques attendues. Dans cette colonne, i symbolise un entier et d un flottant (`double`). Les lettres `S`, `D`, `C`, `L`, `W`, `N` et `M` caractérisent le type d'élément. Enfin, la dernière colonne illustre à quels attributs des classes ces nombres correspondent. Référez-vous aux classes fournies en Annexe A pour identifier les noms des attributs.

Source	S i i i d i	type x y edge alpha wavelength
Cible	D i i i	type x y edge
Cristal	C i i i i	type x y rad mod
Lentille	L i i i i i i	type x y width height wlmin wlmax
Mur	W i i i i	type x1 y1 x2 y2
Bombe	N i i i	type x y rad
Miroir	M i i i i d i i i i d d	type x y length xpad alpha xmin ymin xmax ymax alphamin alphamax

TABLE 1 – Format d'un fichier `.map`

Le code de la Figure 3 fournit un exemple d'une telle carte. Cette carte est garantie sans erreurs, et il existe une solution. Vous constaterez également, au vu des paramètres, qu'il est nécessaire de passer plusieurs fois par le cristal pour faire traverser la lentille au rayon.

Remarque 1. Les miroirs à la ligne 14 et 17 de la carte de la Figure 3 ne sont pas identiques. En effet, ils ne pivotent pas du même côté. Le miroir de la ligne 14 pivote « par la gauche », de l'angle 1.57 radian à l'angle 4.71 radian, alors que le mur de la ligne 17 pivote « par la droite », de l'angle -1.57 radian à l'angle 1.57 radian. Ainsi, même si dans l'absolu, deux angles α_1 et α_2 sont égaux si et seulement si $\alpha_1 \bmod 2\pi = \alpha_2 \bmod 2\pi$, ces angles ne seront pas interprétés de la même façon dans cette application. En particulier, on aura toujours $\alpha_{min} < \alpha_{max}$ dans les spécifications des limites de rotation des miroirs.

5 Exigences et remise de projet

Les points suivants vous sont demandés pour ce jeu. Ne pas remplir l'un de ces objectifs sera systématiquement pénalisé.

```
750 580
S 0 0 29 4.75 400
D 721 275 29
C 464 290 29 40
N 464 232 29
N 464 348 29
L 551 232 58 116 500 600
W 58 0 58 464
W 116 116 116 580
W 174 0 174 464
W 580 0 580 232
W 580 348 580 580
M 0 290 0 58 -1.57 0 0 0 580 -1.57 1.57
M 116 290 0 58 1.57 116 116 116 522 1.57 4.71
M 58 290 0 58 1.57 58 58 58 464 -1.57 1.57
M 174 290 0 58 1.57 174 58 174 464 1.57 4.71
M 116 290 0 58 -1.57 116 116 116 522 -1.57 1.57
M 580 116 0 58 1.57 580 58 580 174 1.57 3.14
M 377 0 0 58 3.14 232 0 522 0 3.14 6.28
M 580 464 0 58 4.71 580 406 580 522 3.14 4.71
M 203 406 29 58 2.35 203 406 290 377 0.78 2.35
M 464 348 29 58 2.35 464 348 551 464 0.78 2.35
M 203 261 29 58 2.35 203 261 290 522 0.78 2.35
```

FIGURE 3 – Exemple de carte

- Fournir une interface graphique permettant de jouer à Starlight en respectant les règles décrites dans ce document et les propriétés de chacun des éléments de la carte.
- L'interface graphique fournie doit être bâtie sur l'architecture « observateur / observé » [5].
- Vous devez être capable de lire et de charger des cartes depuis un fichier texte dont le format est décrit en Section 4.
- Le joueur doit être capable d'allumer et d'éteindre la source. Les rayons sont automatiquement calculés quand la source est allumée. Ils disparaissent quand la source est éteinte.
- Le joueur doit être capable de déplacer et pivoter les miroirs dans les limites autorisées. Si la source est allumée et qu'un miroir est déplacé ou pivoté, les rayons lumineux doivent être automatiquement recalculés.
- Vous devez travailler en binôme. Lors de la défense du projet chaque étudiant du groupe doit être capable de répondre à toute question portant sur quelque partie du code que ce soit.

Vous devez remettre votre projet à votre maître assistant pour le 24 avril 2015 au plus tard.

Des consignes supplémentaires ou alternatives peuvent être formulées par votre maître assistant. Dans tous les cas, elles priment sur celles exprimées ici.

6 Points bonus

Les points suivants sont facultatifs, et sont des suggestions de points à améliorer une fois que vous avez fini les objectifs de base décrits à la Section 5. Fournir certaines de ces fonctionnalités vous rapportera des points bonus. Ces objectifs sont répartis en deux groupes : des objectifs techniques et des objectifs algorithmiques. Dépendant de votre maître assistant, ces objectifs peuvent être cotés différemment.

Objectifs techniques

- Changer la couleur du rayon en fonction de sa longueur d'onde [6].
- Changer l'apparence des rayons lumineux et des éléments de la carte, par exemple via des textures (images).
- Ajouter des effets sonores.
- ☠ Fournir un éditeur de carte « intuitif », par exemple, permettant de placer les composants à la souris. Sa modélisation et conception sont laissées à votre discrétion.

Objectifs algorithmiques

- Permettre de changer les formes de base des éléments du jeu.
- Gérer les collisions entre les miroirs et les autres éléments du jeu. Par exemple, empêcher que l'extrémité des miroirs ne rentre dans les murs.
- ☠ Fournir un générateur de carte aléatoire simple. Remarque : ce point est relativement difficile. N'essayez que si vous avez fini les objectifs secondaires précédents.
- ☠☠ Implémenter un algorithme permettant de résoudre le puzzle automatiquement. Remarque : ce point est *très difficile*. N'essayez que si vous n'avez plus rien d'autre à faire.

Pour rappel, il est inutile d'implémenter un des points ci-dessus tant que vous n'avez pas rempli l'intégralité des objectifs décrits en Section 5.

7 Compléments de géométrie

Cette section rappelle brièvement les notions de trigonométrie et de géométrie qui vous seront utiles à la réalisation de ce projet. Ce rappel est loin d'être complet, vous trouverez plus d'informations relatives à cette discipline sur la page web qui a fortement inspiré cette section [7].

Une façon de décrire les fonctions trigonométriques sinus, cosinus et tangente et d'avoir recours à des triangles rectangles. Cette définition vous sera particulièrement utile lorsque vous voudrez calculer les rayons lumineux.

Étant donné le triangle illustré[5] à la Figure 4, on définit les fonctions trigonométriques de la façon suivante, pour des angles de 0 à $\frac{\pi}{2}$ radians :

$$sin(A) = \frac{a}{c} \tag{1}$$

$$cos(A) = \frac{b}{c} \tag{2}$$

$$tan(A) = \frac{a}{b} \tag{3}$$

Ces fonctions peuvent être inversées via les fonctions asin, acos et atan, disponibles dans la grande majorité des librairies mathématiques, et en particulier dans `cmath` [1,2].

Pour votre projet, vous aurez probablement besoin de calculer la distance entre deux points $P_1 = (x_1, y_1)$ et $P_2 = (x_2, y_2)$. Si tel est le cas, la distance

5. Domaine public

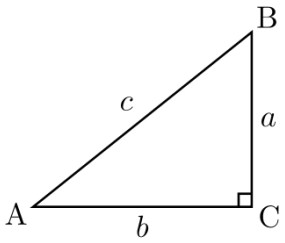

FIGURE 4 – Calcul de fonctions trigonométriques via un triangle rectangle

d entre P_1 et P_2 peut être calculée de la façon suivante [6] :

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

8 Conclusion

Les consignes pour votre projet 2 vous ont été fournies, consistant à implémenter une interface graphique pour jouer à Starlight, sur base d'un squelette de classes. Une liste de points précis à implémenter, ainsi que quelques pistes d'améliorations bonus vous ont également été détaillées.

Vous devez remettre ce projet à votre maître assistant pour le 24 avril 2015 au plus tard.

Références

[1] cmath. http ://www.cplusplus.com/reference/cmath/ - 7 Février 2015.

[2] cmath. http ://en.cppreference.com/w/cpp/header/cmath - 13 mars 2015.

[3] Définition d'un radian. http ://fr.wikipedia.org/wiki/Radian - 7 Février 2015.

[4] Doxygen. http ://www.stack.nl/ dimitri/doxygen/ - 8 Mars 2015.

[5] Observateur / observé. http ://en.wikipedia.org/wiki/Observer_pattern - 8 Mars 2015.

[6] Spectre de lumière visible. http ://fr.wikipedia.org/wiki/Spectre_visible - 7 Février 2015.

6. Vous pouvez facilement trouver cette formule en adaptant la formule de calcul de la longueur de l'hypoténuse d'un triangle rectangle.

[7] Trigonométrie. http ://fr.wikipedia.org/wiki/Trigonométrie - 7 Février 2015.

A Fichiers d'en-têtes

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

/**
 * Cette classe modélise un simple point de coordonées entières,
 * utilisés pour modéliser les positions des objets dans le jeu.
 */
class Point
{
    int x {0};
    int y {0};

  public:
    /**
     * Instancie le point (0,0)
     */
    Point() = default;

    /**
     * Instancie le point de coordonnées spécifiées.
     * @param x l'abscisse du point
     * @param y l'ordonnée du point
     */
    Point(int x, int y);

    /**
     * Retourne l'abscisse du point.
     * @return l'abscisse du point.
     */
    int getX() const;

    /**
     * Retourne l'ordonnée du point.
     * @return l'ordonnée du point.
     */
    int getY() const;

    /**
     * Déplace le point en l'abscisse donnée.
     * @param x l'abscisse où déplacer le point.
     */
    void setX(int x);

    /**
```

```
     * Déplace le point en l'ordonnée donnée.
     * @param y l'ordonnée où déplacer le point.
     */
    void setY(int y);

    /**
     * Déplace le point en la coordonnée donnée.
     * @param x l'abscisse où déplacer le point.
     * @param y l'ordonnée où déplacer le point.
     */
    void setLocation(int x, int y);

    /**
     * Surcharge l'opérateur de flux de sortie pour afficher un
     * récapitulatif des caractéristiques du point sous-jacent en
     * console.
     * @return le flux dans lequel le point a été imprimé.
     */
    friend std::ostream & operator<<(std::ostream & out,
                                     const Point & p);
};

#endif // POINT_H
```

```
#ifndef SOURCE_H
#define SOURCE_H

#include "point.h"
#include <iostream>

/**
 * Modélise la source lumineuse utilisée dans le jeu.
 * </p>
 * La source est un objet carré qui, si allumée, emet un rayon
 * lumineux de longueur d'onde donnée dont l'angle ne peut pas
 * être changé.
 * </p>
 * Le rayon lumineux est émis depuis la position, i.e., le coin
 * supérieur gauche, de la source.
 */
class Source
{
    bool on {false};
    Point pos;
    double alpha;
    int edge;
    int wavelength;

  public:
    /**
     * Instance une nouvelle nource de position, de côté et de
     * longueur d'onde donnée.
     * </p>
     * La position dénote la coordonée du coin supérieur gauche
```

```
     * du carré modélisant la source.
     * </p>
     * La source est intialement éteinte.
     * </p> Si la longueur d'onde du rayon lumineux émis n'est
     * pas comprise entre 360 nm et 830 nm, elle est réglée
     * sur 600 nm.
     * @param p la position de la source.
     * @param e la longueur du côté de la source.
     * @param wl la longueur d'onde du rayon lumineux émis.
     * @see Ray::WL_MIN
     * @see Ray::WL_MAX
     * @see Ray::WL_DFT
     */
    Source(const Point & p, int e, double a, int wl);

    /**
     * Retourne la coordonée du coin supérieur gauche du carré
     * modélisant la destination.
     * @return la coordonée du coin supérieur gauche du carré
     * modélisant la destination.
     */
    const Point & getPosition() const;

    /**
     * Retourne l'anle du rayon émis.
     * @return l'anle du rayon émis.
     */
    int getAngle() const;

    /**
     * Retourne la longueur du côté du carré.
     * @return la longueur du côté du carré.
     */
    int getEdge() const;

    /**
     * Retourne la longueur d'onde du rayon émis.
     * @return la longueur d'onde du rayon émis.
     */
    int getWavelength() const;

    /**
     * Retourne vrai si la source émet un rayon lumineux,
     * faux sinon.
     * @return vrai si la source émet un rayon lumineux,
     * faux sinon.
     */
    bool isOn() const;

    /**
     * Allume ou éteint la source.
     * @param q vrai si la source doit être allumée,
     * faux sinon.
     */
```

```
    void setOn(bool q);

    /**
     * Surcharge l'opérateur de flux de sortie pour afficher
     * un récapitulatif des caractéristiques de la source
     * sous-jacente en console.
     * @return le flux dans lequel la source a été imprimée.
     */
    friend std::ostream & operator<<(std::ostream & out,
                                     const Source & s);
};

#endif // SOURCE_H
```

```
#ifndef DEST_H
#define DEST_H

#include "point.h"

#include <ostream>

/**
 * Cette classe modélise la destination utilisée dans le jeu.
 * </p>
 * Une destination est un objet carré qui, quand traversé par
 * un rayon limuneux, fait remporter la partie au joueur.
 */
class Dest
{
    Point pos;
    int edge;
    bool light {false};

  public:
    /**
     * Intancie une destination, de position et rayon donné.
     * @param p le coin supérieur gauche du carré modélisant
     *        la destination.
     * @param e la longueur du côté du carré.
     */
    Dest(const Point & p, int e);

    /**
     * Retourne la position du coin supérieur gauche du carré
     * modélisant la destination.
     * @return la position de la destination.
     */
    const Point & getPosition() const;

    /**
     * Retourne la longueur du côté du carré.
     * @return la longueur du côté du carré.
     */
    int getEdge() const;
```

```
    /**
     * Retourne vrai si la destination est illuminée,
     * faux sinon.
     * @return vrai si la destination est illuminée,
     * faux sinon.
     */
    bool isLightedUp() const;

    /**
     * Illumine la destination ou non.
     * @param vrai si la destination doit être illuminée,
     * faux sinon.
     */
    void setLightedUp(const bool q);

    /**
     * Surcharge l'opérateur de flux de sortie pour afficher
     * un récapitulatif des caractéristiques de la destination
     * sous-jacente en console.
     * @return le flux dans lequel la destination a été imprimée.
     */
    friend std::ostream & operator<<(std::ostream & out,
                                     const Dest & s);
};

#endif // DEST_H
```

```
#ifndef CRYSTAL_H
#define CRYSTAL_H

#include "point.h"
#include <ostream>

/**
 * Cette classe modélise les cristaux utilisés dans le jeu.
 * </p>
 * Un cristal est un objet circulaire centré en un point, et
 * d'un certain rayon.
 * </p>
 * Un rayon lumineux passant à travers un crystal modifie sa
 * longueur d'onde (en l'augmentant ou en la diminuant d'une
 * certaine valeur) mais pas sa trajectoire.
 */
class Crystal
{
    Point center;
    int rad;
    int mod;

  public:
    /**
     * Instancie un cristal centré au point donné, d'un certain
     * rayon et modifiant la longueur d'onde des rayons qui le
```

```
     * traversent d'une valeur donnée.
     * @param p le centre du cristal
     * @param r le rayon du cristal
     * @param m le modificateur de longueur d'onde du cristal
     */
    Crystal(const Point & p, int r, int m);

    /**
     * Retourne la coordonée du centre du cristal
     * @return  la coordonée du centre du cristal
     */
    const Point & getCenter() const;

    /**
     * Retourne le modificateur de longueur d'onde du cristal
     * @return  le modificateur de longueur d'onde du cristal
     */
    int getModifier() const;

    /**
     * Retourne le rayon du cristal
     * @return le rayon du cristal
     */
    int getRadius() const;

    /**
     * Surcharge l'opérateur de flux de sortie pour afficher
     * un récapitulatif des caractéristiques du cristal
     * sous-jacent en console.
     * @return le flux dans lequel le cristal a été imprimé.
     */
    friend std::ostream & operator<<(std::ostream &,
                                     const Crystal &);

};

#endif // CRYSTAL_H
```

```
#ifndef LENS_H
#define LENS_H

#include "point.h"

#include <ostream>

/**
 * Cette classe modélise les lentilles utilisées dans le jeu.
 * </p>
 * Une lentille est un objet rectangulaire qui ne laisse passer
 * les rayons lumineux que dans un certain intervalle de longueur
 * d'onde. Si un rayon lumineux se trouve dans l'intervalle de
 * longueur d'onde autorisé, il traverse la lentille sans subir
 * aucune modification. Sinon, la lentille se comporte comme un
 * mur.
```

```
 */
class Lens
{
    Point pos;

    int width;
    int height;

    int wlmin;
    int wlmax;

  public:
    /**
     * Instancie une lentille à l'aide de toutes ses
     * caractéristiques.
     * @param p la position du coin supérieur gauche du
     *          rectangle modélisant la lentille.
     * @param w la largeur de la lentille
     * @param h la hauteur de la lentille
     * @param wlmin la longueur d'onde minimale des rayons
     *              autorisés à franchir la lentille
     * @param wlmax la longueur d'onde maximale des rayons
     *              autorisés à franchir la lentille
     */
    Lens(const Point & p, int w, int h, int wlmin, int wlmax);

    /**
     * Retourne la position du coin supérieur gauche du
     * rectangle modélisant la lentille.
     * @return la position du coin supérieur gauche du
     * rectangle modélisant la lentille.
     */
    const Point & getPosition() const;

    /**
     * Retourne la largeur de la lentille
     * @return la largeur de la lentille
     */
    int getWidth() const;

    /**
     * Retourne la hauteur de la lentille
     * @return la hauteur de la lentille
     */
    int getHeight() const;

    /**
     * Retourne la longueur d'onde minimale des rayons
     * autorisés à franchir la lentille
     * @return la longueur d'onde minimale des rayons
     * autorisés à franchir la lentille
     */
    int getMinWavelength() const;
```

```
    /**
     * Retourne la longueur d'onde maximale des rayons
     * autorisés à franchir la lentille
     * @return la longueur d'onde maximale des rayons
     * autorisés à franchir la lentille
     */
    int getMaxWavelength() const;

    /**
     * Surcharge l'opérateur de flux de sortie pour afficher
     * un récapitulatif des caractéristiques de la lentille
     * sous-jacente en console.
     * @return le flux dans lequel la lentille a été imprimée.
     */
    friend std::ostream & operator<<(std::ostream & out,
                                     const Lens & m);
};

#endif // LENS_H
```

```
#ifndef WALL_H
#define WALL_H

#include "point.h"
#include <iostream>

/**
 * Cette classe modélise les murs utilisés dans le jeu.
 * </p>
 * Les murs sont des segments de droite qui ne réfléchissent
 * pas la lumière.
 */
class Wall
{
    Point start;
    Point end;

  public:
    /**
     * Instancie un mur.
     * @param p1 le début du mur.
     * @param p2 la fin du mur.
     */
    Wall(const Point & p1, const Point & p2);

    /**
     * Retourne le début du mur.
     * @return le début du mur.
     */
    const Point &getStart() const;

    /**
     * Retourne la fin du mur.
     * @return la fin du mur.
```

```
     */
    const Point & getEnd() const;

    /**
     * Surcharge l'opérateur de flux de sortie pour afficher
     * un récapitulatif des caractéristiques du mur sous-jacent
     * en console.
     * @return le flux dans lequel le mur a été imprimé.
     */
    friend std::ostream & operator<<(std::ostream &,
                                     const Wall &);
};

#endif // WALL_H
```

```
#ifndef NUKE_H
#define NUKE_H

#include "point.h"
#include <ostream>

/**
 * Cette classe modélise les bombes utilisées dans le jeu.
 * </p>
 * Une bomnbe est un objet circulaire qui, si illuminé par
 * un rayon, fait perdre la partie au joueur.
 */
class Nuke
{
    Point pos;
    int rad;
    bool light {false};

  public:
    /**
     * Instancie une bombe en une position donnée avec un rayon
     * déterminé.
     * @param p la position de la bombe
     * @param r le rayon de la bombe
     */
    Nuke(const Point & p, int r);

    /**
     * Retourne la position de la bombe.
     * @return la position de la bombe.
     */
    const Point & getLocation() const;

    /**
     * Retourne le rayon de la bombe.
     * @return le rayon de la bombe.
     */
    int getRadius() const;
```

```
    /**
     * Retourne vrai si la bombe est illuminée, faux sinon.
     * @return vrai si la bombe est illuminée, faux sinon.
     */
    bool isLightedUp() const;

    /**
     * Illumine la bombe ou non.
     * @param q vrai si la bombe est illuminée, faux sinon.
     */
    void setLightedUp(bool q);

    /**
     * Surcharge l'opérateur de flux de sortie pour afficher un
     * récapitulatif des caractéristiques de la bombe
     * sous-jacente en console.
     * @return le flux dans lequel la bombe a été imprimée.
     */
    friend std::ostream & operator<<(std::ostream & out,
                                     const Nuke & s);
};

#endif // NUKE_H
```

```
#ifndef MIRROR_H
#define MIRROR_H

#include "point.h"

#include <ostream>

/**
 * Cette classe modélise les miroirs utilisés dans le jeu.
 * </p>
 * Un miroir est un segment de droite dont la propriété est
 * de réfléchir la lumière d'un seul côté uniquement. Si un
 * rayon lumineux touche un miroir du côté non réfléchissant,
 * le miroir se comporte comme un mur.
 * </p>
 * Les miroirs sont capables d'être déplacés et pivotés dans
 *  une certaine limite.
 */
class Mirror
{
    Point pivot;
    int length;
    int xpad;
    int xMin {0};
    int xMax {0};
    int yMin {0};
    int yMax {0};
    double alpha;
    double alphaMin {0};
    double alphaMax {0};
```

```
public:
  /**
   * Instancie un miroir en une position donnée, d'une certaine
   * longueur et orienté d'un certain angle.
   * </p>
   * Comme dans ce constructeur les limites de déplacement et
   * de rotation du miroir ne sont pas définies, ce miroir
   * peut se déplacer et pivoter librement.
   * @param p la position (et le point de pivot) du miroir
   * @param len la longueur du miroir
   * @param x le décallage du pivot par rapport au bord gauche
   *          du miroir
   * @param a l'angle d'inclinaison du miroir
   */
  Mirror(const Point & p, int len, int x, double a);

  /**
   * Instancie un miroir en une position donnée, d'une certaine
   * longueur et orienté d'un certain angle.
   * </p>
   * Ce constructeur permet également aux miroirs de pivoter
   * dans une certaine limite.
   * </p>
   * Si l'intervalle de limite de déplacement (e.g., sur les
   * abscisses) [a,b] est tel que a = b, le miroir ne peut
   * être déplacé sur l'axe considéré.
   * </p>
   * Si l'intervalle de limite d'inclinaison [a,b] est tel que
   * a < b, le miroir pivote dans le sens horloger, si a = b le
   * miroir ne peut pas pivoter, si a > b, le miroir
   * pivote dans le sens anti-horloger.
   * @param p la position (et le point de pivot) du miroir
   * @param len la longueur du miroir
   * @param x le décallage du pivot par rapport au bord gauche
   *          du miroir
   * @param a l'angle d'inclinaison du miroir
   * @param min l'abscisse et l'ordonnée minimum du miroir.
   * @param max l'abscisse et l'ordonnée maximum du miroir.
   * @param amin l'angle d'inclinaison minimum du miroir.
   * @param amax l'angle d'inclinaison maximum du miroir.
   */
  Mirror(const Point & p, int len, int x, double a, Point min,
         Point max, double amin, double amax);

  /**
   * Retourne la position (et le pivot) du miroir.
   * @return la position (et le pivot) du miroir.
   */
  const Point & getPivot() const;

  /**
   * Retourne la longueur du miroir
   * @return la longueur du miroir
```

```
     */
    int getLength() const;

    /**
     * Retourne le décallage du pivot par rapport au bord gauche
     * du miroir.
     * @return le décallage du pivot par rapport au bord gauche
     * du miroir.
     */
    int getXPad() const;

    /**
     * Retourne l'inclinaison du miroir.
     * @return l'inclinaison du miroir.
     */
    double getAngle() const;

    /**
     * Retourne l'inclinaison minimum du miroir.
     * </p>
     * Si l'intervalle de limite d'inclinaison [a,b] est tel que
     * a < b, le miroir pivote dans le sens horloger, si a = b
     * le miroir ne peut pas pivoter, si a > b, le miroir pivote
     * dans le sens anti-horloger. Si a = b = 0, le miroir peut
     * être pivoté librement.
     * @return l'inclinaison minimum du miroir.
     */
    double getMinAngle() const;

    /**
     * Retourne l'inclinaison maximum du miroir.
     * </p>
     * Si l'intervalle de limite d'inclinaison [a,b] est tel
     * que a < b, le miroir pivote dans le sens horloger, si
     * a = b le miroir ne peut pas pivoter, si a > b, le miroir
     * pivote dans le sens anti-horloger. Si a = b = 0, le miroir
     * peut être pivoté librement.
     * @return l'inclinaison minimum du miroir.
     */
    double getMaxAngle() const;

    /**
     * Retourne la position minimum du miroir.
     * </p>
     * Si l'intervalle de limite de déplacement (e.g., sur les
     * abscisses) [a,b] est tel que a = b, le miroir ne peut
     * être déplacé sur l'axe considéré. Si a = b = 0, le miroir
     * peut être déplacé librement.
     * @return la position minimum du miroir.
     */
    Point getMinPivot() const;

    /**
     * Retourne la position maximum du miroir.
```

```
     * </p>
     * Si l'intervalle de limite de déplacement (e.g., sur les
     * abscisses) [a,b] est tel que a = b, le miroir ne peut
     * être déplacé sur l'axe considéré. Si a = b = 0, le miroir
     * peut être déplacé librement.
     * @return la position minimum du miroir.
     */
    Point getMaxPivot() const;

    /**
     * Déplace le miroir en la position donnée, si c'est
     * autorisé. Retourne vrai si le déplacement a été
     * effectué correctement, retourne faux sinon.
     * @return vrai si le déplacement a été effectué
     * correctement, retourne faux sinon.
     * @see Mirror::getPivot()
     */
    bool setPivot(const Point &);

    /**
     * Pivote le miroir sur un angle donné, si c'est
     * autorisé. Retourne vrai si la rotation a été effectuée
     * correctement, retourne faux sinon.
     * @return vrai si la rotation a été effectuée
     * correctement, retourne faux sinon.
     * @see Mirror::getAngle()
     */
    bool setAngle(double);

    /**
     * Retoune vrai si le miroir peut être pivoté sur
     * l'angle donné, retourne faux sinon.
     * @return vrai si le miroir peut être pivoté sur
     * l'angle donné, retourne faux sinon.
     * @see Mirror::getAngle()
     */
    bool checkAngleRange(double) const;

    /**
     * Retoune vrai si le miroir peut être éplacé en la
     * position donnée, retourne faux sinon.
     * @return vrai si le miroir peut être déplacé en
     * la position donnée, retourne faux sinon.
     * @see Mirror::getPivot()
     */
    bool checkPivotRange(const Point &) const;

    /**
     * Surcharge l'opérateur de flux de sortie pour
     * afficher un récapitulatif des caractéristiques du
     * miroir sous-jacent en console.
     * @return le flux dans lequel le miroir a été imprimé.
     */
    friend std::ostream & operator<<(std::ostream & out,
```

```
                                        const Mirror & m);
};

#endif // MIRROR_H
```